UNIVERSIDADE FEDERAL FLUMINENSE

Daniel Eduardo Teles de Almeida

AVALIAÇÃO DE USO DE AUTOMAÇÃO DE TESTES

Niterói

2019

Daniel Eduardo Teles de Almeida

AVALIAÇÃO DE USO DE AUTOMAÇÃO DE TESTES

Trabalho de Conclusão de Curso submetido ao Curso de Tecnologia em Sistemas de Computação da Universidade Federal Fluminense como requisito parcial para obtenção do título de Tecnólogo em Sistemas de Computação.

Orientador:
Flavio Luiz Seixas

Niterói
2019

Ficha catalográfica automática - SDC/BEE
Gerada com informações fornecidas pelo autor

T269a Teles de almeida, DANIEL EDUARDO
AVALIAÇÃO DE USO DE AUTOMAÇÃO DE TESTES / DANIEL EDUARDO Teles de almeida ; Flavio Luiz Seixas, orientador. Niterói, 2019.
60 f. : il.

Trabalho de Conclusão de Curso (Graduação em Tecnologia de Sistemas de Computação)-Universidade Federal Fluminense, Instituto de Computação, Niterói, 2019.

1. Teste. 2. Automação. 3. Gestão de Projeto. 4. Qualidade. 5. Produção intelectual. I. Seixas, Flavio Luiz, orientador. II. Universidade Federal Fluminense. Instituto de Computação. III. Título.

CDD -

Bibliotecária responsável: Fabiana Menezes Santos da Silva - CRB7/5274

DANIEL EDUARDO TELES DE ALMEIDA

AVALIAÇÃO DE USO DE AUTOMAÇÃO DE TESTES

Trabalho de Conclusão de Curso submetido ao Curso de Tecnologia em Sistemas de Computação da Universidade Federal Fluminense como requisito parcial para obtenção do título de Tecnólogo em Sistemas de Computação.

Niterói, 04 de junho de 2019.

Banca Examinadora:

_____

Prof. Flavio Luiz Seixas, Orientador. – UFF - Universidade Federal Fluminense

_____

Prof. Altobelli de Brito Mantuan, Avaliador. – UFF - Universidade Federal Fluminense

Dedico este trabalho a minha esposa, minha mãe, meu amigo Felipe que não me deixou desistir e a todos os amigos e professores que me ajudaram a chegar até aqui.

AGRADECIMENTOS

À meu Orientador Flavio Seixas pela paciência e por não desistir de mim.

À meus companheiros de profissão por compartilharem suas experiencias.

À todos os meus familiares e amigos pelo apoio e colaboração.

RESUMO

O benefício principal da automação de testes de sistemas no ciclo de vida de desenvolvimento é a manutenção da qualidade do software entregue e a economia de tempo e esforço de testes. Como consequência, a descoberta e resolução de bugs e outros problemas de desenvolvimento é acelerada agilizando o tempo de projeto. Esta pratica gera economia com qualidade e também minimiza o fator fadiga de um trabalho muitas vezes repetitivo. Analisaremos onde a automação de sistemas geraria benefícios que compensam seu custo de desenvolvimento e tempo de projeto. Também onde existe expectativa de manutenção por meio da inclusão de novas funcionalidades e correção de falhas de um sistema que geraria a necessidade de constante de execução de regressões. Além da avaliação do tempo de projeto em relação ao de desenvolvimento das ferramentas de automação de forma a justificar a implantação da mesma. Posteriormente serão vistos também alguns casos práticos onde a automação aumentou a velocidade dos testes e garantiu qualidade na entrega. Finalmente, além de avaliar as melhorias obtidas com os métodos estudados, onde foi observado a encurtamento entre as implantações de novas versões em produção, e a redução dos tempos de teste, acelerando a busca e correção de falhas nos sistemas analisados e melhorando a visão do cliente em relação a qualidade e velocidade da entrega. Devemos entender que a automação não pretende substituir o trabalho humano, simplesmente aumentar sua eficiência, já que um julgamento humano em cima da usabilidade é indispensável e que pode ser o diferencial entre o cliente considerar um projeto como bem-sucedido ou desastroso.

Palavras-chaves: Automação, Testes, Qualidade e Objetivos.

ABSTRACT

The Main benefit when considering system testing automation in the development life-cycle is the maintenance of quality delivered and the testing time and effort. This generates economy and hastens the discovery and resolution of bugs and other development problems as consequence and shortens a projects time. Those practices generate economy with quality, and also minimizes the tiring factor of a work that is in many occasions repetitive. An Analysis will be done as to where System testing automation would generate benefits that would make up its development cost and the project required time. Also, in some cases where there is the expectation of future maintaining by including new features and bug fixes generating the constant need to execute regression testing. The evaluation of the benefit of using project's time to develop the automated scenarios and tools in order to justify its implementation while guaranteeing its quality. Afterwards, a study on a couple of real cases where automation improved testing speed and guaranteed a delivery with more quality will be done. Finally, beyond looking at the improvements generated by the studied methods, where more versions were deployed to production more often, and testing speed improvements leading to faster bug-finding and bug-fixing as well as the improved customer's perception on the software quality and delivery time. It must be understood that automation is not meant to replace human labor, only to improve its efficiency, since a human judgement on usability is irreplaceable and might be the differential between the project being successful or a disaster in the customer's view.

Keywords: Automation, Testing, Quality e Objectives.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

API	Application Programming Interface - Interface de programação para aplicações
REST	Representational State Transfer
SOAP	Simple Object Access Protocol
POS	Point of Sale ou Point of Service
WSDL	Web Services Description Language
APF	Análise de Ponto de Função
OS	Operating System ou Sistema Operacional.
OTA	Over the Air
PO	Product Owner.
APK	Android Package
AWS	Amazon Web Services
SWEBOK	Software Engineering Body of Knowledge
ISTQB	International Software Testing Qualifications Board
MPT	Melhoria do Processo de Testes Brasileiro
TMMI	Testing Maturity Model Integration
MPS	Melhoria de Software Brasileiro
CMMI	Testing Maturity Model Integration
PIB	Produto Interno Bruto
TDD	Test Driven Development
RC	Release Candidate
IDE	Integrated Development Environment
QR-CODE	Quick Response Code
DSL	Domain Specific Language
UI	User Interface

SUMÁRIO

1 INTRODUÇÃO

Durante o ciclo de vida de desenvolvimento de *softwares*, independentemente da sua metodologia, existem diversas etapas. Uma dessas etapas é aquela onde ocorrem testes de qualidade e desempenho na busca por eventuais *bugs* e outros erros de funcionamento. A varredura de um sistema para análise da qualidade de *software* e prevenção da entrada de erros em produção é trabalho do analista de sistemas, independente do foco na área de qualidade.

Esperar melhorias é comum quando adequamos processos completamente manuais para tarefas automatizadas, entretanto, o trabalho da automação, além de economia de tempo e potencialização de qualidade, é incrementar a performance do profissional que terá tempo para investigações mais complexas e não-mecânicas ao se considerar a subjetividade de algumas análises. Procura-se mostrar qual o melhor ponto de equilíbrio para se desenvolver um produto de qualidade.

De Acordo com o SWEBOK 3.0, guia do corpo de conhecimento para engenharia de software que descreve práticas de engenharia de software amplamente aceitas [1]. Qualidade de *software* é uma área de conhecimento que podem se referir tanto as características desejadas de um produto de software quanto as ferramentas e processos existentes para garantir tais características.

Neste trabalho, buscamos maneiras de se garantir uma maior qualidade na entrega de um projeto de *software*, utilizando a automação para este fim. Os Projetos de *software* constantemente sofrem atrasos ou entregam menos valor do que o esperado devido a deficiência de qualidade que poderia ser evitada, ou ao menos mitigada com a implantação de alguns procedimentos. Quanto mais cedo iniciados na fase de desenvolvimento e testes de um projeto, mais economia geram durante sua implantação em produção. Isto é uma consequência já reconhecida no campo acadêmico e empresarial, além do incremento da percepção de qualidade gerada para o usuário final

e a melhoria da relação desta organização com a que é responsável pelo desenvolvimento [2, pp. 27-31].

Espera-se que através da implantação de ferramentas de automação e testes automatizados integrados, unitários e instrumentados com execução a cada nova alteração no código, com o auxílio de ferramentas de integração contínua, se tenha alcançado a qualidade necessária para entrega do produto, além de uma análise de quando será necessária a execução de testes manuais. O objetivo final é termos um processo que derive uma entrega de projeto mais rápida, com menos atrasos e imprevistos e com mais qualidade.

Este trabalho tem como objetivo avaliar os benefícios para qualidade do desenvolvimento e utilização de ferramentas de automação em testes de *software*. Os testes de *software*, ainda são uma parte muito negligenciada em muitos projetos, apesar de termos visto uma clara evolução na importância dada para a atividade.

Os custos de desenvolvimento caem com os testes, e vemos uma queda maior ainda em alguns casos com a implantação de uma automação. Analisaremos o ganho que as automações trazem aos projetos, em termos de qualidade, financeiros e de tempo, visto que o assunto está intrinsecamente ligado a área de testes, estudando seu histórico, vantagens e estado atual.

Um levantamento será feito das ferramentas existentes atualmente para a automação de plataformas web, Android, *back-end*, além da descrição de ferramentas criadas especificamente para propósitos de escopo único, como geração de massa e execução automatizada de *scripts* com verificação de resultados, assim como as linguagens utilizadas em cada estudo para que o leitor possa entender os estudos de caso que seguem.

Analisaremos situações onde foi observado aumento no engajamento da equipe no uso de automações e sua consequente economia de tempo e esforço mesmo em tarefas mais simples, onde originalmente levaria muito tempo, e seu consequente

impacto no planejamento de futuros projetos que passou a considerar o uso das ferramentas e automações desenvolvidas.

Serão feitos dois estudos de caso com a finalidade de entender um pouco mais como podemos ganhar qualidade independentemente da metodologia de desenvolvimento aplicada, ao se utilizar ferramentas que garantam uma manutenção da qualidade a cada alteração, garantindo a equipe de testes uma maior cobertura e liberando o analista responsável para trabalhos mais intelectuais e menos mecânicos e repetitivos.

Dentro de um grande projeto, os testes são uma atividade essencial no ciclo de desenvolvimento de sistemas. Devemos considerar que humanos cometem erros, seja por vício na validação do próprio código, seja por falta de experiência, ou por não ter ciência da funcionalidade integrada no sistema fora do escopo do próprio desenvolvimento [3, p. 226].

Buscar qualidade na entrega de produto é sempre uma preocupação das empresas. Onde e quando investir recursos para esse fim? Quais os procedimentos trazem melhores resultados com menos custo? Ilustrar a ideia de que automatizar 100% do campo de testes traz resultados que interferem positivamente na qualidade de entrega do produto indicando a necessidade do auxílio de testes humanos ou ponderamentos referentes ao gasto do processo de testes. Procurando mostrar qual o melhor ponto de equilíbrio para se desenvolver um produto de qualidade.

Através de uma análise comparativa de estudo de casos espera-se promover uma avaliação qualitativa acerca da automação do ambiente de testes. Se trata de reunir informações relativas ao processo de automação nessa etapa de desenvolvimento de softwares, analisando eventuais desvios e imprevistos nesse trabalho além de revelar as intervenções humanas necessárias durante o processo de automação. Por fim busca-se concretizar ideias sobre a busca por qualidade nesse processo de automação com a

intenção de entender até que ponto a automação é eficiente e traz qualidade para o processo de testes.

A estrutura deste trabalho estará dividida inicialmente em um capítulo descrevendo a importância dos testes no desenvolvimento de *software*, seus procedimentos e além dos níveis de testes e a importância de automatizá-los. Posteriormente, realizamos a comparação na atuação desta fase de um projeto em modelos ágeis e em cascata, para então formar a base que utilizaremos para o capítulo onde serão descritos dois casos de estudo, em duas empresas com perfis e metodologias distintas, demonstrando a diferença de performance e qualidade de uma série de atividades considerando o uso de automações contra o teste manual, suas implicações no perfil do profissional, na gestão do projeto e manutenção da qualidade.

2 REFERENCIAL TEÓRICO

Este capítulo apresentará conceitos relativos aos testes e melhoria de qualidade por intermédio de automações e boas práticas. A Seção 2.1 apresentaremos alguns padrões do mercado em relação a boas práticas de processos aplicados na engenharia de *software*, incluindo algumas específicas a definição, construção e execução de testes de *software*.

A Seção 2.2 tem relação com o custo associado a atividade de testes, com uma análise do momento atual, além de motivações para se investir em testes e a consequências de não o fazer. Também é levado em consideração o custo de tempo em relação a cobertura desejada. Na 2.3 avalia-se as técnicas de desenvolvimento de casos de teste, além da percepção de complexidade associada que tem como potencial consequência na estimativa da atividade.

Na Seção 2.4 iremos avaliar as fases de teste e como se enquadram em relação ao ciclo de vida de desenvolvimento de *software*, bem como em relação a sua implantação e consequentemente o tipo comumente associado. Na 2.5 será feita uma breve análise dos motivos para se automatizar um sistema. A Seção 2.6 apresenta técnicas para a realização dos testes.

As Seções 2.7 e 2.8 são reservadas a apresentação de algumas ferramentas de gestão e controle para melhor planejamento da execução das atividades no ciclo de vida de um sistema e as ferramentas utilizadas para se automatizar alguns tipos de teste respectivamente. Na Seção 2.9 apresentamos o conceito de integração contínua e como sua implantação pode contribuir para a manutenção da qualidade de um projeto. Finalmente será realizada uma comparação de onde se enquadram as atividades de teste nas metodologias Ágil e Cascata na Seção 2.10.

2.1 PADRÕES DE MERCADO REFERENTES A PRÁTICAS DE QUALIDADE DE SOFTWARE

Ao se falar em testes, é importante termos como guias os padrões do mercado, a ISO29119[1] e o *Syllabus* do ISTQB - *International Software Testing Qualifications Board*[2], tais guias devem ser utilizados por todos os profissionais que desejam trabalhar na área de qualidade de *software* e seus focos são na estruturação e execução de testes, porém também temos outros documentos com foco em gestão e controle como o Melhoria do Processo de Testes Brasileiro MPT.br[3] e o *Testing Maturity Model Integration* TMMI[4] que diverge do MPS.br principalmente no custo de implementação, mas com as mesmas práticas. Outros modelos que também fazem parte do tema, convivendo com os citados anteriormente são o Melhoria de *Software* Brasileiro - MPS.br[5] e o CMMI - *Capability Maturity Model Integration*[6] . Tais modelos exigem um investimento de tempo e monetário, não só para sua implementação, mas também para a recertificação, porém definitivamente contribuem para a maturidade do processo das empresas que realizam tais investimentos.

2.2 – INVESTIMENTO EM TESTES

O custo de verificação de *software* em muitas vezes ultrapassa a metade do custo total do desenvolvimento e manutenção do mesmo [4], mas se deve considerar que de forma genérica, o nível de investimento varia de acordo com certas considerações como a complexidade e foco do sistema sendo desenvolvido.

---

1 http://softwaretestingstandard.org/
2 https://www.istqb.org/downloads/syllabi/foundation-level-syllabus.html
3 http://mpt.org.br/
4 https://www.tmmi.org/about-tmmi-assessment-method-tam/
5 https://softex.br/mpsbr/
6 https://cmmiinstitute.com/

Os Testes são uma etapa de extrema importância visto que tem por objetivo validar se a aplicação é funcional e atende aos requisitos da especificação. Os testes podem ser realizados utilizando-se diferentes técnicas e em diferentes etapas do projeto.

Dentre as razões que temos para um investimento adequado em testes, está na qualidade e confiabilidade do produto entregue ao cliente, que a experiência do usuário na usabilidade do *software* seja adequada, e que o mesmo seja seguro [3, pp. 37-39]. Além disso, o custo da resolução de um problema aumenta exponencialmente em relação ao ciclo de vida de projeto que se encontra, podendo chegar a ser 100 vezes mais caro de corrigir na fase de operação do que a de testes conforme podemos ver na Figura 1. Em 2002 foi realizado um estudo nos estados unidos, que avaliou que *bugs* de *software* geraram um prejuízo equivalente a 0.6% do PIB americano [5].

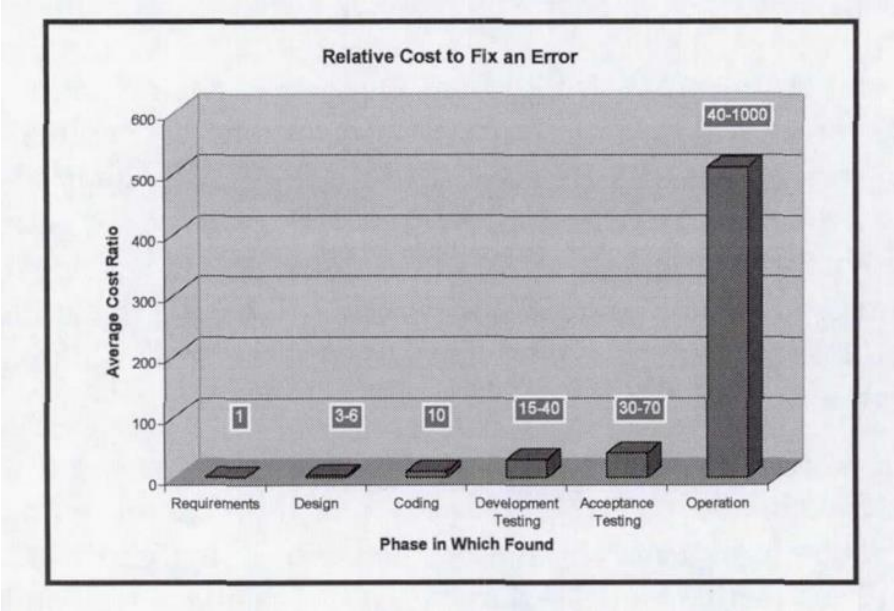

Figura 1: Custo relativo a fase de desenvolvimento na correção de erros [6]

Para se verificar a cobertura de teste de acordo com a complexidade, devemos determinar qual o percentual de cobertura desejamos atingir de acordo com a maturidade

do *software*. A criação de casos de teste para todas as possibilidades em um software não é prático [7, p. 10]. Uma versão inicial de um teste pode ter uma abrangência grande, porém pouco estressado em pontos específicos. Já uma versão mais madura de um *software*, pode se dedicar a aumentar a sua cobertura de testes, para que se possa verificar com maior nível de detalhe funcionalidades específicas que podem ter sido negligenciadas anteriormente. O objetivo a ser atingido deve ser a maximização do número de erros que possam ser encontrados em relação a um número finito de casos de teste [7, p. 13].

O conhecimento da atividade e suas limitações é fundamental nessa atividade, pois alguns elementos que tem a sua cobertura desejada podem não ser executáveis e a decisão em cima do que se deve desconsiderar no catálogo de casos de teste deve envolver a decisão do testador.

No que se refere ao foco, verificamos que se o produto desenvolvido possuir poucas funcionalidades, como um módulo que faz parte de um grande sistema, seu teste individual tende a não necessitar de muitos casos para uma boa cobertura da qualidade de sua implementação, principalmente se a integração dos diferentes módulos e os testes de integração necessários para esta fase não forem de responsabilidade da mesma empresa, no entanto, dependendo da quantidade de funcionalidades implementadas, a cobertura pode ser igualmente grande mesmo antes de sua integração.

2.3 AVALIAÇÃO DA ESCRITA DOS CENÁRIOS DE TESTES E SEU ESFORÇO ASSOCIADO.

Existem diversas formas de se avaliar como se desenvolver os cenários de teste de um dado sistema. Serão destacadas algumas das práticas mais comuns, e é importante notar que é comum que mais de uma técnica seja usada, seja por conveniência ou para que se tenha maior assertividade.

Os Pontos de Função se utilizam das métricas de esforço de desenvolvimento e se baseando nisso, é calculado o esforço e tempo alocado a testes. Existe discordância sobre a análise por pontos de função ser uma medida objetiva [8]. Isto é baseado no fato de a contagem não ser de fato uma medida, porém uma avaliação, e, portanto, subjetiva. Existem também sugestões de especialistas para adotar critérios extraídos da literatura científica que permita não realizar uma análise subjetiva da técnica de APF [9].

Nos Pontos de Testes, além das métricas extraídas no ponto de função, utilizamos na estimativa apresentada, fatores como ambientes de desenvolvimento/testes, conhecimento dos profissionais envolvidos, qualidade e quantidade de documentação disponível entre outros. Exemplos de fatores que podem alterar a estimativa são a performance do ambiente de testes, que afetaria diretamente na velocidade de execução dos mesmos, assim como o conhecimento da equipe de testes a respeito da regra de negócios e a metodologia de testes utilizada.

Em Casos de Uso, baseado nos eventos possíveis de utilização do sistema, desenvolvemos os cenários de testes identificados e seus possíveis resultados esperados, calculando a partir daí o esforço de testes necessário. O responsável por realizar essa estimativa, deve avaliar não só a quantidade de casos a serem realizadas, mas também a complexidade individual. Um simples teste de entrada e saída de dados não exige o mesmo esforço do que um teste ponta a ponta de um sistema com múltiplas integrações entre módulos.

Complexidade das Telas, envolve o desenvolvimento dos cenários de testes e estimativa de seu tempo de execução baseados na complexidade das telas do sistema.

Devemos avaliar todas as entradas possíveis e caminhos alternativos passíveis de geração de acordo com as opções disponíveis.

Requisitos, nesta técnica, a escrita dos casos de teste é baseada na documentação de requisito. Similarmente às outras estimativas, se avaliam as possíveis alternativas de caminhos a se seguir no programa e sua complexidade, mas desta vez sem ter como parâmetro um produto pronto, e sim a descrição dos requerimentos para o desenvolvimento do mesmo e os caminhos do ponto de vista do usuário ou arquiteto que criou o documento.

Histórias de Usuários: baseado nas descrições da visão apresentada pelos usuários a respeito do *software* a ser desenvolvido, criamos os cenários de testes. Existem diversos formatos para a criação de histórias de usuários, onde se identifica um papel, uma meta e uma consequência, e a partir daí a desenvolvemos. Ao criar uma história de usuário para um cenário hipotético de realização de cadastro em uma rede social teríamos: Eu como usuário, desejo criar uma conta na rede social, para ter acesso a mesma.

2.4 TIPOS DE TESTE E SUAS FASES

Com os possíveis cenários de testes em mãos, é preciso entender quais as técnicas mais apropriadas a se utilizar para a realização dos mesmos, além do nível de conhecimento o usuário possui do código por trás do sistema e o acesso ao mesmo para a melhor técnica para realização do mesmo.

O **Teste unitário** também conhecido como caixa branca, ou estrutural, necessita de conhecimento profundo da estrutura do sistema. Tem como objetivo testar as unidades do código fonte, que podem ser os métodos ou classes de um objeto [2, p. 148]. Em muitos casos os objetos testados possuem dependências que podem não ter

sido desenvolvidas, ou que a sua utilização acarretaria em atrasos nos testes. Para esses casos se utilizam *mockobjects*, que são objetos que simulam um comportamento e operações e simular retornos do programa específicos a casos de testes que somente ocorreriam em exceções [2, p. 149].

É necessário um bom planejamento de testes que cubra boa parte do código-fonte para evitar comprometer a qualidade do mesmo. Uma maneira de avaliar os casos de teste necessários é se utilizando da complexidade ciclomática, que corresponde a quantidade de caminhos independentes do código [10].

Este tipo de teste é muito utilizado na metodologia de desenvolvimento **TDD** - *Test-Driven Development*, que se baseia em desenvolver testes que falhem e posteriormente realizar a codificação do sistema de maneira que os testes unitários passem e posteriormente com o prosseguimento do desenvolvimento e refatoração permaneçam com o resultado esperado [2, p. 155]. Em muitos casos os desenvolvedores investem uma quantidade considerável de tempo a escrever testes unitários, até por questão de orgulho de seu código.

Nos **testes de integração** devemos verificar a integração entre os diferentes componentes de *software* que compõem o sistema e tem como objetivo encontrar falhas na integração interna dos componentes do sistema, e normalmente os problemas encontrados nesta fase são de falhas de transmissão de dados [2, pp. 153-154], como por exemplo, um componente estar esperando um tipo de objeto ou valor como *output* de um outro componente e receber um tipo ou valor não esperado. Importante notar, que de acordo com Somerville só existem 3 tipos de testes: Unidade, Integração e de sistema [2, pp. 20-21].

Testes funcionais são realizados de acordo com a especificação do cliente para cada funcionalidade de uma aplicação. Ele se enquadra em um **teste de caixa preta**, onde não é necessário conhecimento do código-fonte para sua realização, somente as saídas que devem ser geradas para uma dada entrada [3, p. 35] e se o programa se

comporta conforme o esperado [7, p. 13]. Um exemplo de teste funcional, seria testar as saídas de um *webservice* usando o SOAP-UI[7] para diversas entradas, sem necessariamente estarem conectadas ao sistema ao qual o uso do *webservice* é pretendido. Já os **testes não-funcionais** servem para testar aspectos não funcionais do sistema, como performance e confiabilidade. Em alguns casos existem testes não-funcionais que são efetivamente o aspecto mais importante de um sistema. Um sistema bancário, por exemplo, tem como maior prioridade a segurança, pela própria natureza do produto oferecido. Já um sistema que controla o acesso a entrada de um grande evento, necessita que sua integridade esteja inabalável, preferencialmente com sistemas de redundância como plano de emergência.

Outro teste de caixa preta é o **teste de aceitação**. O mesmo deve ser executado em um ambiente espelho do que será usado no final, durante a implantação do sistema, por um representante da organização de desenvolvimento junto a um representante da organização do usuário final [11] [2, p. 162]. Em muitos casos se realizam testes *Alpha*, Beta e Gama. Os mesmos são definidos baseados no estágio do desenvolvimento do *software* e do tamanho do grupo que realizará o teste. Este depende do teste, onde os testes *Alpha* tem um pequeno grupo de usuários que podem incluir família e amigos ou o resto do time de desenvolvimento no local de desenvolvimento, o beta, onde já é exposto a um número de usuários, que pode ser grande e não mais restrito [2, p. 160] e o gama.

Visto que as versões betas de aplicativos ainda estão em desenvolvimento, normalmente ainda existem muitas falhas e os desenvolvedores aproveitam o maior número de usuários para registrar mais falhas com eficácia. Os testes beta funcionam em algumas ocasiões também como um tipo de marketing [2, p. 160]. Algumas grandes empresas como o Google costumam realizar esse tipo de teste [12]. Já nos testes gama, *release candidate* [13] ou ***RC***, ou Golden Master como é chamado pela Apple, o produto é disponibilizado para o usuário final que reporta as falhas, caso as encontre.

[7] https://www.soapui.org/

Em alguns casos, após o teste de aceitação realizado com os usuários para validação funcional, temos também o **teste de aceite operacional**, realizado para se validar aspectos não-funcionais do sistema antes de sua implementação no ambiente final. O mesmo deve ser realizado, assim como os testes de aceitação do usuário, em um ambiente o mais próximo do final o possível, e tem como objetivo garantir a confiabilidade e performance do sistema

Após a entrega do produto, é comum no ciclo de vida de um *software* que mudanças ocorram, as mesmas podem ser necessárias devido a novas funcionalidades ou mesmo correção de erros descobertos após a sua implementação, justamente o que o estudo conduzido neste trabalho pretende evitar com o aumento da qualidade proporcionado pela automação e maturidade profissional. Entretanto, quando é necessária de uma nova versão do *software* com alterações, um novo teste é necessário. Além dos testes específicos inerentes às alterações realizadas, também se faz necessário a realização dos testes das outras funcionalidades já testadas anteriormente, devido a chance de introdução de novos erros causados durante a alteração no código devido a atividade de desenvolvimento ser uma atividade realizada por humanos. A realização destes testes de funcionalidades já testadas em versões anteriores é chamada de **teste de regressão**.

2.5 MOTIVOS PARA AUTOMATIZAR

A atividade de testes dentro de um projeto, é essencial a economia e manutenção da qualidade do produto entregue a um cliente durante o ciclo de desenvolvimento, juntamente aos diversos testes manuais que devem ser executados. Entretanto a automação também ajuda ainda mais nos quesitos apresentados, visto que tarefas repetitivas podem ser tediosas e passíveis de erro [3, pp. 97-100]. Um teste de repetição para avaliar um *memory leak,* por exemplo, seria trabalhoso e desgastante para se

realizar manualmente, onde uma simples automação de repetição iria resolver o problema.

Para avaliar a viabilidade de um projeto de automação, devemos levar em consideração cinco fatores [3, p. 97]: Seleção de ferramentas de teste existentes, se disponível; possibilidade e custo de construção de ferramentas específicas para automação de testes; disponibilidade de treinamento do usuário para essas ferramentas e tempo / esforço necessário; custo total, incluindo custos para aquisição de ferramentas, suporte, treinamento e uso destas; impacto no recurso, cronograma e gerenciamento do projeto.

Sabe-se que nem todos os tipos de testes devem ser automatizados, principalmente quando se referem a um crivo humano para avaliação de percepção de qualidade [3, p. 249], o que não significa que não possamos usar uma infinidade de ferramentas para o auxílio de testes manuais, como por exemplo, um gerador de massa de testes. Essas ferramentas são úteis para aumentar a eficiência e eficácia dos testes [14, pp. 24-26]. Devemos, então, verificar os principais tipos de testes que se beneficiariam de uma avaliação.

Dois tipos de testes que mais teriam benefícios ao ter uma abordagem direcionada a automação são: testes de desempenho, que exigem repetição ou múltiplas conexões simultâneas e testes de Estresse, para verificar os pontos de ruptura de uma execução sob recursos limitados [15].

2.6 TÉCNICAS PARA REALIZAÇÃO DE TESTES

Geralmente, a execução dos testes segue de um ponto de partida, com algum dado sendo alimentado para um sistema, e sua saída sendo avaliado de acordo com critérios previamente estabelecidos. Quando o resultado da execução de algum teste

não vai de acordo com o esperado, temos um *bug*. Ou seja, uma falha no processo de execução de dado programa. A cobertura dos diversos tipos de testes citados anteriormente, garante uma maior confiabilidade ao sistema, porém, a experiência do testador conta para que sejam considerados casos mais incomuns, mas que não seriam impossíveis de se ocorrer na mão de um usuário comum, como por exemplo, um cenário onde se deve testar o comportamento da aplicação na troca entre a conexão *wifi* e outras disponíveis e onde a solução encontrada poderia ser mover fisicamente o dispositivo para longe do roteador.

Dentre as técnicas mais comuns estão a entrada de dados fora dos limites estabelecidos, além dos limites máximos e mínimos permitidos, tipos de dados incorretos, e casos de teste de exceção, como uma chamada realizada entre dois dias distintos de uma mudança de ciclo de faturamento em um sistema de telefonia.

Em alguns casos a organização demandante já desenvolve um roteiro de testes a ser seguido, com uma validação mais próxima de mecânica do que intelectual. Nestes casos o profissional se sobressai ao conhecer a implementação do código, ao auxiliar o desenvolvedor do produto a descobrir a falha com celeridade.

2.7 FERRAMENTAS DE GESTÃO E CONTROLE

O acompanhamento da gestão das atividades de implementação de novas funcionalidades no sistema sendo desenvolvidas ou *features* e *bugs* ou falhas em um programa [14, p. 64] encontrados é vital para uma rápida resolução e priorização dos problemas. Algumas ferramentas online auxiliam no acompanhamento destas questões. É essencial para equipes que não trabalham no mesmo ambiente e têm, portanto, comunicação limitada. Entre algumas vou destacar quatro dessas ferramentas que são: Jira, Mantis, Test Collab e Test Link.

O Jira é usado em diversas empresas, tendo benefícios como relatórios, organização de sprints para organizações que utilizam o método ágil, Kanban Board, que é um auxílio visual das atividades em um quadro, dividido em uma série de raias para melhor visualização das tarefas e seus estados atuais e nesta ferramenta é usado para facilitação da alteração de *status* das *issues*, que são as atividades a serem realizadas, divisão das atividades entre os membros da equipe e comunicação entre a equipe sobre um mesmo problema de forma direcionada com manutenção de histórico e armazenamento de artefatos, como as evidências de *bugs* encontrados ou documentação compartilhada entre a equipe no desenvolvimento de um tipo de uma atividade específica[8].

A Mantis é uma ferramenta simples com *open-source* e grátis para controle de *bugs*[9]. Sua integração com ferramentas de controle de versão, possibilita verificar os repositórios do projeto, além de visualização de histórico das *issues*, uma Seção de wiki e o *roadmap*, um plano estratégico que para que se atinja um objetivo e inclui marcos definidos para isso, que possibilita acompanhar o andamento de algum conjunto específico de *issues*.

Test Collab é mais focado na gestão da equipe de teste, de cenários abertos e realizados e relatórios baseados nestas informações. Possui integração com ferramentas de gestão como o Jira para abertura de *issues* e associação de casos de testes a *issues* abertas[10].

Test Link Ferramenta grátis e *open-source* para gestão do processo de testes, que assim como o Test Collab tem integração com ferramentas onde são realizados o registro de *bugs* como o Jira e o Mantis e pode ser customizado visual ou funcionalmente[11].

---

[8] https://br.atlassian.com/software/jira
[9] https://www.mantisbt.org/
[10] https://testcollab.com/
[11] http://testlink.org/

A escolha da ferramenta deve ser pensada levando-se em consideração o tamanho do projeto e o poder aquisitivo da empresa tocando o mesmo. O Jira é uma ferramenta que possui a maioria das funcionalidades necessárias em um só lugar, porém, principalmente em equipes maiores tem um custo significativo. As ferramentas de código aberto podem suprir as necessidades se utilizadas em conjunto e justamente pelo fato de ser de código aberto tem como possibilidade alterações na sua implementação de acordo com as necessidades de quem o usa.

2.8 FERRAMENTAS DE AUTOMATIZAÇÃO

Para se automatizar algum teste, devemos enquadrá-lo em algum dos tipos de teste, dependendo de sua plataforma, e avaliar se existe alguma ferramenta que possa facilitar a maneira de se realizar o trabalho. Em alguns casos, pode ser necessário a construção de uma ferramenta própria. No caso de testes unitários, as IDEs já criam a estrutura de projeto para o desenvolvimento dos mesmos e quando se deseja testar as classes desacopladamente, sem depender de algum objeto da resposta de um outro método, usamos *mocks*, que são objetos que substituem o código original emulando a funcionalidade real para que se possa avaliar o funcionamento do comportamento do código [16, pp. 287-301]. Alguns dos *frameworks* que se podem citar para a geração de *mocks* em testes unitários em Android, são o mockito[12] e o mockk para utilização com kotlin [13].

Webservice é uma tecnologia baseada em XML e HTTP que servem para disponibilizar serviços para outras aplicações. A troca de mensagens entre as aplicações e o *webservice* utiliza o protocolo SOAP e seus serviços são expostos através de um XML chamado *Web Services Description Language* ou WSDL que informam a maneira

[12] https://site.mockito.org/
[13] https://mockk.io/

com a qual o serviço deve ser invocado, quais os parâmetros de entrada, os tipos de dados, suas descrições e localização. Em muitos momentos é necessário se realizar os testes de tais APIs, *webservices* ou serviços REST, utilizados para integração entre sistemas de maneira desacoplada. Para automatizarmos tais testes, podemos utilizar o SOAP-UI. A ferramenta possibilita testes automatizados funcionais, testes de segurança, de performance além da geração de relatórios dos testes realizados.

Em aplicações *web* como fornecedores de serviços, tais quais bancos, provedores de mídia, redes sociais*, e-commerce*, sistemas de gestão de inventário na nuvem, entre outros, temos o Selenium, que automatiza *browsers* de internet, portanto é usado primariamente para automação de aplicações *web* com propósitos de testes, mas não limitado somente a isso. Tarefas administrativas também podem se aproveitar do poder da ferramenta para serem automatizadas[14].O Selenium pode ser usado em conjunto com diversas IDEs como Eclipse e Visual Studio e em diversas linguagens de programação como JAVA, C#, Ruby, Python e Javascript[15].

Entrando no mundo dos testes funcionais de aplicativos para dispositivos móveis, temos o Appium, desenvolvido para realizar testes automatizados em dispositivos móveis como Tablets e Celulares, sendo uma evolução do Selenium pois usa o mesmo protocolo, mas com adições para se trabalhar com estes dispositivos, já que eles fazem muito mais do que um *browser*. Assim como o Selenium ele trabalha com diversas IDEs em diversas linguagens[16].

[14] https://www.seleniumhq.org/
[15] https://www.seleniumhq.org/download/
[16] http://appium.io/docs/en/about-appium/getting-started/

Tais opções cobrem parte dos tipos de automações para os serviços da maioria das empresas e na maioria dos casos não só a execução dos testes pode ser realizada de forma automatizada, como também a execução da automação que realiza tais testes.

2.9 INTEGRAÇÃO CONTÍNUA

Os testes realizados a cada nova entrega de uma versão de *software*, podem já estar automatizados, e em muitos casos, tais testes automatizados já garantem uma confiabilidade na continuidade da qualidade alcançada anteriormente. Entretanto, a execução dos mesmos se enquadra em um dos processos que, caso sejam realizados manualmente, são passíveis de erros humanos em sua execução, ou podem ser acidentalmente ignorados durante um período particularmente conturbado de uma entrega com um prazo apertado. Para que possamos evitar esse tipo de ocorrência, ao mesmo tempo que garantimos uma maior eficiência, podemos contar com a integração contínua.

Integração contínua consiste em algo simples: a integração da alteração do código dos desenvolvedores ao projeto principal com frequência, em detrimento de se realizar somente uma vez e tem como objetivo avaliar se as alterações ou funcionalidades não criaram novos defeitos no projeto existente.

A cada alteração do código possui um processo automatizado que passa por diversas etapas, como a execução dos testes automatizados integrados ao código da aplicação e o *build* ou construção do código executável da aplicação. Esta abordagem leva a uma detecção mais rápida e frequente de erros de integração e ajuda a reduzir significativamente problemas que só seriam detectados em fases futuras [17].

Algumas ferramentas para integração contínua automatizada podem ser utilizadas como o Jenkins ou o Hudson, ambas gratuitas. Jenkins é um servidor de automação *open-source* que pode ser usado para automatizar todos os tipos de tarefas

relacionadas à construção, teste e entrega ou *deploy* de *software*[17]. Tanto o Jenkins quanto o Hudson suportam ferramentas de controle de versão como o GIT, e podem executar *scripts* shell ou batch de Windows.

2.10 FASE DE TESTES NAS METODOLOGIAS CASCATA E ÁGIL

Os ciclos de vida de desenvolvimento de sistemas têm como objetivo produzir conteúdo de alta qualidade que vá de encontro ou supere a expectativa do cliente. Dentre as diversas metodologias de desenvolvimento, normalmente alinhadas com os padrões do mercado, exigências do cliente e filosofia de desenvolvimento da empresa de desenvolvimento, destacamos a cascata e a ágil.

O modelo de desenvolvimento em cascata foi originado em indústrias de manufaturamento e construção e possui este nome pois o progresso do projeto sempre flui em uma direção, como uma cachoeira [18].

Em sua implementação original, o modelo em cascata continha as seguintes fases em ordem: requerimento, análise, *design*, construção ou codificação, teste e Operação (que envolve instalação, migração, suporte e manutenção dos sistemas completos).

O modelo em cascata precisa que a fase anterior esteja revisada e verificada antes que a próxima possa iniciar [19], dada a ordenação intrínseca a utilização deste modelo, a equipe de testes é em muitas vezes subutilizada até que a fase de testes de fato se inicie. Uma de suas vantagens é a grande ênfase que é colocada na documentação, e as equipes podem se utilizar das fases iniciais para já planejarem as atividades e automações possíveis a serem implementadas futuramente, sobretudo de testes regressivos em projetos em cima de sistemas que já estão bem estabelecidos e estáveis,

[17] https://jenkins.io/doc/

onde usualmente o modelo em cascata se encaixa melhor, ou em projetos grandes na qual não sejam esperadas grandes mudanças de escopo [3, pp. 231-232].

Na metodologia ágil por outro lado, tem como objetivo uma entrega de valor mais rápido e com maior frequência para o cliente, com maior ênfase na maleabilidade das entregas, já que as mesmas podem ser repriorizadas, tanto em escopo, como em prazo e uma menor em documentação. Para muitas empresas, que operam em um ambiente com mudanças constantes que impossibilita requisitos estáveis, fazem a escolha consciente de focar em entregas rápidas em detrimento de qualidade e requisitos bem definidos [2, p. 38]. Usualmente o *framework* SCRUM é indicado para projetos ágeis. O mesmo é composto por ciclos de desenvolvimento, conhecidos como *sprints,* curtos em duração [2, p. 50] e pouco adaptativos. No ágil, a atividade de testes é realizada em todas as fases do projeto, validando os requisitos desde a sua criação até a entrega final. Para tal, os testadores devem entender as *user Stories*, que são divisões na forma de incrementos funcionais do trabalho a ser realizado [20], escrever os casos de teste, que é uma amostra dos tipos de entradas na execução de uma aplicação e suas saídas esperadas [7, p. 32], preparar o ambiente para o teste e finalmente, executá-los. Os testes exploratórios são de suma importância, visto que o curto período dos *sprints* pode levar algum caso específico a não ser considerado.

A automação na metodologia ágil deve ser realizada e mantida por toda a equipe, mas idealmente é realizada pela equipe de testes que pode realizar a automação em paralelo com o desenvolvimento das histórias no produto [21, pp. 307-315]. No entanto, o desenvolvimento das mesmas deve avaliar se o custo (tempo) de desenvolvimento não ultrapassa a utilidade da mesma (tempo de execução manual / quantidade de vezes que a mesma será executada na prática). Além de uma avaliação periódica no catálogo de testes automatizados de maneira a reduzir o inchaço desnecessário dos casos de teste, por repetição ou irrelevância dos mesmos com a evolução do produto, e também para evitar que a regressão automatizada seja um gargalo ao se consolidar alterações [22].

Até o atual ponto do trabalho, foram apresentados alguns conceitos sobre testes sistêmicos, sua cobertura, o desenvolvimento de seus casos de teste, o que são cada tipo de teste, suas particularidades, algumas das ferramentas usadas no mercado para automatizá-los e os motivos para fazê-lo. Obviamente existem muitas nuances que podem ser aprofundadas no caso de uma especialização, entretanto já está formada a base necessária ao entendimento dos casos de estudo apresentados a seguir.

3 ESTUDOS DE CASO

Analisaremos situações onde foi observado aumento no engajamento da equipe no uso e desenvolvimento de automações, tanto por meio de relatórios que as ferramentas desenvolvidas internamente registravam no banco de dados para metrificação de casos de teste executados. Estas ferramentas foram desenvolvidas internamente pois as disponíveis comercialmente não atendiam totalmente as necessidades da equipe ou não atendiam com a mesma velocidade e eficiência. O maior engajamento da equipe também foi observado pelo aumento do numero de cenários automatizados que passaram a entregues por diferentes membros da equipe e com maior frequência.

Também avaliaremos a consequente economia de tempo e esforço mesmo em tarefas mais simples, onde originalmente se tomaria muito tempo de execução, e seu consequente impacto no planejamento de futuros projetos que passou a considerar o uso das ferramentas e automações desenvolvidas.

A implementação de tais ferramentas e processos automatizados serão avaliadas na Seção 3.1 em um projeto que ocorria em uma empresa de telecomunicações cujo projeto de desenvolvimento segue a metodologia em cascata e utilizou primariamente ferramentas desenvolvidas internamente assim como o caso de uma empresa que utiliza a metodologia ágil e passou a realizar testes automatizados em conjunto com integração contínua em diversos níveis de testes para seu *software* de *Smart-POS* na Seção 3.2.

3.1 CASO 1 - A EMPRESA DE TELECOMUNICAÇÕES

Esta empresa do ramo de telecomunicações, uma das líderes no país, tem, a cada projeto realizado para seu já estabelecido sistema de gerenciamento de negócio,

independentemente de sua área usuária demandante, dentre elas, novos planos e promoções, *marketing*, redes, tarifação e faturamento, geração de relatórios, manutenção de clientes/base e suporte a produção, existe a necessidade de alguns passos básicos na geração da massa de testes e seu processamento para a avaliação dos casos de teste, onde em muitas vezes o processo produtivo se torna ineficiente para reprodução em tempo de desenvolvimento.

A metodologia utilizada pela empresa é em cascata. Um documento inicial é rascunhado pela área responsável pela demanda, repassada ao especialista que desenvolve tecnicamente a solicitação e é enviada para a empresa fornecedora da atualização de *software*, que incidentalmente, que analisa o documento, desenvolve de acordo e entrega a solicitante, que então repassa o pacote entregue para outra empresa que realiza os testes de integração e de aceitação junto ao usuário final que é responsável pela aprovação formal que seja autoriza sua entrada em produção.

Dado uma demanda onde seria necessário testar a alteração de tarifa de um determinado plano, o procedimento a ser tomado desconsiderando qualquer ferramenta, seria utilizar o *front-end* utilizado nas lojas onde um cliente assinaria um contrato de serviços, onde precisaríamos preencher os campos obrigatórios com nome, endereço, estado, telefone, e-mail, data de nascimento conforme Figuras 2 e 3 e posteriormente, em uma segunda tela onde escolheríamos o plano do cliente e seus serviços associados, buscaríamos e associar um número de telefone produtivo, ou seja, que esteja na rede real de telecomunicação em uma faixa reservada para testes, além do mesmo procedimento para o número lógico do *chip* que você estiver utilizando no aparelho conforme representado nas Figuras 4 e 5.

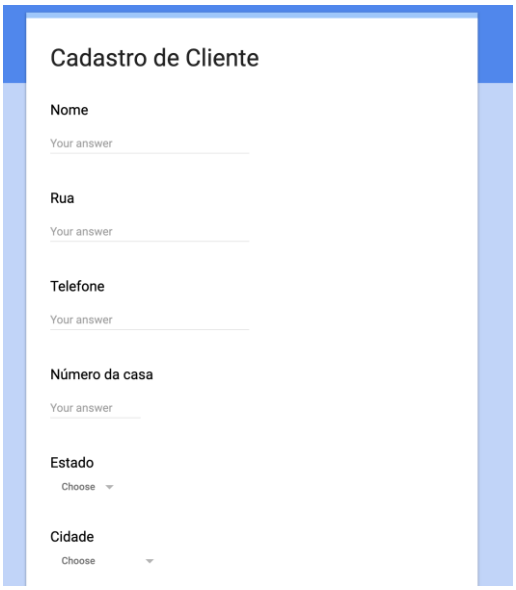

Figura 2: Representação de tela do front-end de cadastro de cliente da operadora parte 1

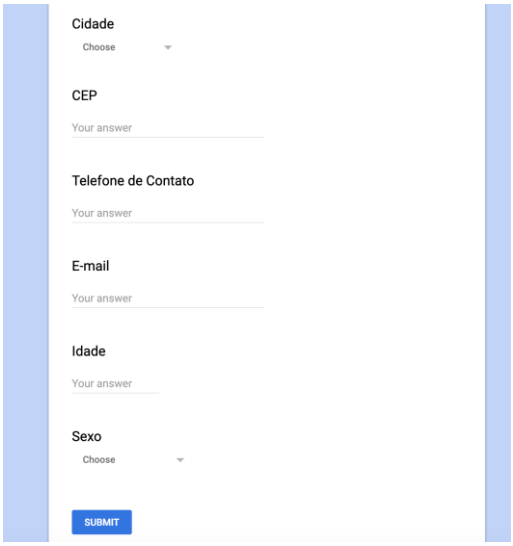

Figura 3: Representação da tela de cadastro de uma operadora parte 2

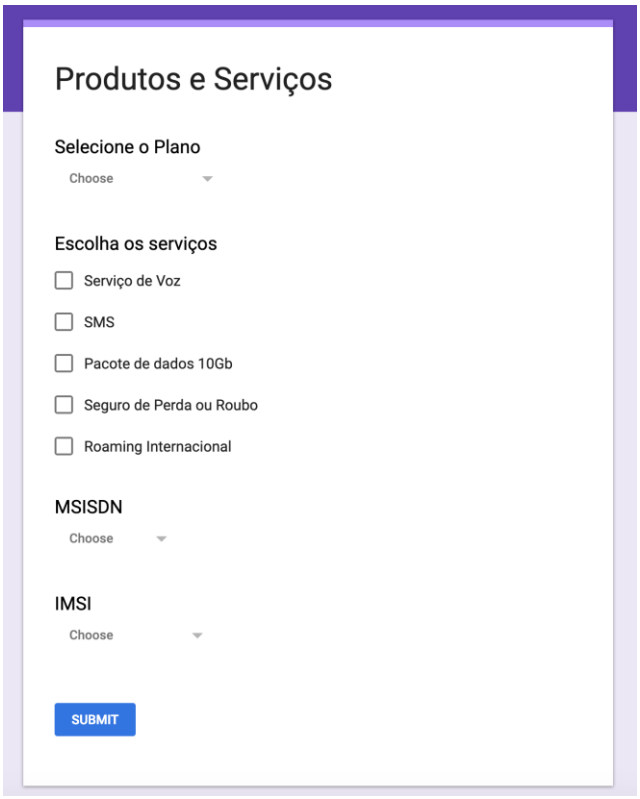

Figura 4: Representação da tela de seleção de produtos e serviços de uma operadora

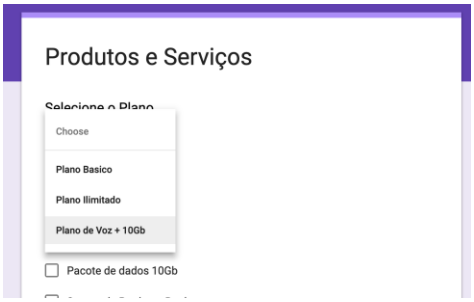

Figura 5: Demonstração do *drop-down* da tela acima

Depois, devemos aguardar o procedimento que roda em *back-end*, realizar uma chamada telefônica real em um aparelho celular, verificar nos servidores da mediação, o arquivo com os bilhetes das chamadas realizadas anteriormente. Servidor este onde são disponibilizados os arquivos com os metadados da chamada, necessários à verificação de regra de tarifação, executar os cerca de 6 módulos da cadeia de tarifação, cada um com a sua responsabilidade específica, como verificar a consistência das informações do bilhete, o valor da chamada, a existência algum tipo de promoção ou

cobrança diferenciada para o seu cliente entre outros processos. Posteriormente, seria necessário rodar os módulos de faturamento, que tem como objetivo gerar o XML que serve de entrada para o processo de geração da fatura em PDF que seria enviada ao cliente, ou para impressão e envio para o dito cliente, além da validação de outras regras referentes a pacotes que podem ser descontados em tempo de tarifação ou de faturamento, somente então seria possível validar em um cenário próximo ao de produção a alteração da tarifa de fato ao visualizar a alteração na fatura.

Se formos contabilizar o tempo necessário ao desenvolvimento do teste supracitado, avaliaríamos o tempo médio de preenchimento de um campo multiplicado pelo número de campos existentes no front-end. Se considerarmos a velocidade média de digitação de um indivíduo de 33 palavras por minuto [23] e considerarmos que um formulário de cadastro possui 11 campos (nome, sobrenome, rua, número da casa, informações adicionais apartamento ou referência, cidade, estado, cep, telefone de contato, e-mail, idade, sexo) com uma média de 2 palavras por campo mais 2 segundos para troca entre campos, seria necessário 1 minuto e 2 segundos para a entrada dos dados, mais 5 segundos para a busca e seleção do plano na *combobox* além de outros 20 segundos (tempo de sistema) para a execução do processo de associação de número com mais 20 para o de associação do número lógico do chip, cerca de 1 dia inteiro para rodar o processo de tarifação (pela quantidade de dados no banco de testes) e outro dia inteiro para rodar os módulos da cadeia de faturamento teríamos 2 dias rodando módulos além de 1 minuto e 47s digitando dados para a geração de massa.

Além do excesso de trabalho gerado pela demora do processo, que exigia que um profissional estivesse sempre no aguardo do término da execução dos processos para a validação do cenário perto do fim do prazo, também haviam outros efeitos colaterais gerados, como uma base de dados de testes mais inconsistente, devido as intervenções dos testadores com o objetivo de tentar amenizar a demora das consultas realizadas pelos módulos em um ambiente, que mesmo com 10% dos dados de produção, devido ao *hardware* menos potente, demoravam horas para serem finalizadas. Esse excesso de trabalho obviamente gerava insatisfação e criava uma política corporativa que

valorizava o profissional menos pela sua capacidade e mais pela sua resiliência durante projetos mais críticos.

Para agilizar o processo de testes descrito no cenário acima, primeiramente foi criada internamente uma ferramenta para geração rápida da massa de testes, os clientes fictícios, que realizariam as chamadas. Esta ferramenta solicitava como únicas entradas, o código de área e o código do plano necessário a criação da massa de testes. No cenário da verificação de uma alteração de tarifa, o nome, endereço entre outros dados não importam, então poderíamos usar simplesmente dados fixos ou a partir de uma tabela de dados adaptáveis selecionados com base no estado do plano, visto que em alguns casos, impostos e benefícios locais podem alterar a tarifa de um plano similar. Ainda no quesito geração de massa, como benefício da geração automática de massa, com o auxílio de outra ferramenta de automação, é gerado o arquivo que contém os dados da ligação, visto que a chamada telefônica a partir de um número real não só seria um procedimento desnecessário, já que o teste não pretende testar a resiliência da rede durante uma chamada, como passível de erros, visto que para uma chamada possua o tempo exato de ligação necessária a validação de dada regra tarifária, contaria com a possibilidade de atraso na precisão em sua execução devido a comunicação com as torres celulares, para a realização de um teste que não tem como objetivo validar essa parte do processo, que é desacoplada do sistema e não sofreu alterações. Tal gerador de clientes do exemplo anterior também acelera a execução dos testes, por somente se comunicar com os serviços responsáveis por criar as informações do cliente no banco de dados e não na rede de telecomunicações.

O processo de criação de massa tomaria cerca de 2 segundos para a escolha da opção após a execução do Shell Script representado na Figura 6, 2 segundos para o preenchimento da área do cliente a ser criada DDD 21 para um cliente do RJ por exemplo e mais 2 segundos para a digitação do código do plano, anteriormente fornecido em uma lista conforme representação na Figura 7, buscada no banco usando a informação anteriormente capturada para filtrar os planos que fossem elegíveis a área que o cliente criado para o teste supostamente residiria, o XML que o *front-end*, no nosso cenário

demorado, montaria para os serviços de *back-end* são então criados com essas informações da ferramenta interna de automação e enviadas para o servidor responsável por esse processo, que nos retornam no XML de resposta as informações da massa de dados, mostradas de maneira formatada na tela para o usuário cuja representação do código se encontra na Figura 8, em um processo que demora menos de 10 segundos no seu total.

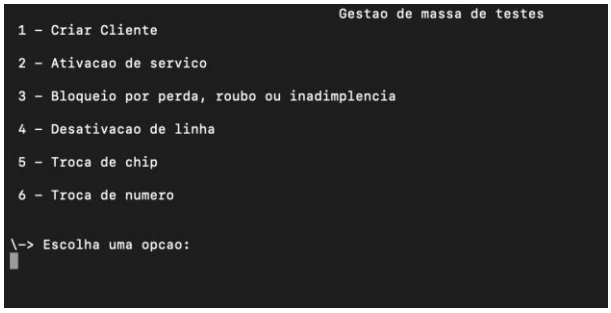

Figura 6: Representação da tela de um script em Shell/Bash destinado a geração rápida de massa de testes para o cenário descrito

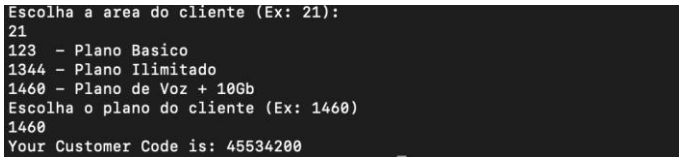

Figura 7: Representação da tela seguinte, ao se escolher a primeira opção para criação de cliente

```
. functions.sh

echo "Escolha a area do cliente (Ex: 21):"
read vCustomerArea

#This will show available customer plan lists
F_EXECUTE_SQL_QUERY(verify_available_plans_by_area.sql $vCustomerArea)

echo "Escolha o plano do cliente (Ex: 1460)"
read vCustomerPlan

vCustomerCreationXml=F_EXECUTE_SQL(create_customer_xml.sql $vCustomerArea $vCustomerPlan)

vXmlResponse=F_SEND_XML($vCustomerCreationXml)
vCustomerCode=F_MAPXML($vXmlResponse "customer_code")

echo "Your Customer Code is: $vCustomerCode"
```

Figura 8: Trecho de código de exemplo de output da tela acima escrito em shell script

O Selenium, descrito anteriormente na Seção 2.8 poderia ser usado para o preenchimento dos dados do formulário de criação de clientes de forma automática, e a

automação foi desenvolvida e está disponível para uso, nos casos onde o teste envolve alterações no sistema *front-end*, ainda assim comparando a performance da criação da massa de testes usando o Selenium, com a ferramenta criada internamente, é possível notar uma melhoria em relação ao processo manual, mas ainda um atraso substancial, já que a automação teria que preencher os dados do formulário, mesmo que mais rápido que um humano, e aguardar a troca de mensagens entre os servidores para múltiplos processos, nem todos necessários a realização da maioria dos testes.

Estando o cliente referente a massa de teste criado pela ferramenta de automação em mãos, podemos gerar a chamada para ele, que com uma outra ferramenta que tem como entrada o código do cliente gerado juntamente aos dados do tipo de chamada desejada para o teste, e como *output*, a ferramenta já cria o arquivo com os dados da ligação no diretório onde será processado o bilhete da chamada na execução dos módulos de tarifação, execução esta, que será realizada utilizando a terceira ferramenta interna que cria um ambiente propício a execução dos módulos de tarifação e faturamento de maneira acelerada para finalidade de testes ao criar *views on-demand* com os dados de teste, evitando que os módulos precisassem varrer todo o banco de dados réplica de produção. Tal otimização acelera a execução dos módulos de tarifação para menos de 10 minutos e ao utilizar a mesma técnica para as tabelas do banco de dados de faturamento, temos redução similar. Posteriormente a ferramenta é responsável por retornar aplicação ao estado original similar ao produtivo.

Com isso encurtamos um teste que potencialmente demoraria um pouco mais de dois dias, para um processo que nos tomaria aproximadamente meia hora, permitindo adaptabilidade em sua execução, já que anteriormente, a melhor técnica a ser utilizada era a preparação da massa de testes de todos os cenários e posteriormente uma única execução, necessitando esforço similar no caso da necessidade de retrabalho, e após a melhoria, passou a ser possível testar cenários em menores quantidades para avaliação individual de comportamento e maior celeridade na necessidade de retrabalho. Os profissionais tiveram mais espaço para que pudessem implementar novas melhorias, como o desenvolvimento de outra ferramenta que já realiza uma avaliação completa da cadeia descrita em todos os distintos casos do catálogo de regressão de dado módulo, com intervenção somente para a avaliação do relatório gerado pelo script que realizava

a execução de todos os processos automatizados melhorados em ordem. A lógica da execução de tal ferramenta era similar a execução de testes unitários e instrumentados nas IDEs de programação, com uma avaliação de sucesso ou falha de acordo com um critério pré-programado que já alertava ao usuário da ferramenta onde se encontram os pontos de falha, na avaliação do teste realizado, caso existissem. A aplicação de tais ferramentas também possibilitou o desenvolvimento de outras capacidades nos profissionais, que tiveram evoluções distintas, pois passaram a ser avaliados com base em quesitos mais técnicos.

3.2 CASO 2 - A EMPRESA DE PAGAMENTOS

Uma empresa do ramo de pagamentos necessita testar lançamentos de versões do OS ou atualizações de aplicativos de seu *smart-POS*, baseado em Android. O dispositivo já é utilizado em diversos estabelecimentos por todo o brasil, e seu *software* já está estabilizado, no entanto, é desenvolvido para um mercado que sempre está mudando com adições como pagamento via *QR-CODE* ou *Contactless*, com a utilização de cartões por aproximação, ou até mesmo de *smartphones* e *smartwatches*, cartões de benefícios com saldo pré-pago para alimentação/refeição e combustível, assim como alterações das agências regulatórias em relação a diversos itens como as informações que devem ser apresentadas em um comprovante de vendas, e mensagens informativas que devem ser apresentadas na tela. a cada alteração de seu principal aplicativo representado na Figura 9, que é responsável pela realização de transações financeiras e pela comunicação com o *host* de pagamento, deve ocorrer uma certificação, exigida pelas bandeiras de cartão com o intuito de garantir a confiabilidade das transações financeiras realizadas através do POS. A aposta comercial da empresa é que o dispositivo, por usar Android, atraia estabelecimentos que desejem customizar a sua experiência de vendas com o desenvolvimento de aplicativos próprios, para uso exclusivo, ou distribuição em loja virtual própria, assim como os grandes *players* do mercado de *smartphones*. Dentre os exemplos de aplicativos existentes, pode-se

destacar um que integre a cozinha de um restaurante ao garçom, onde além facilitar os pagamentos, agiliza a preparação dos pedidos dos clientes.

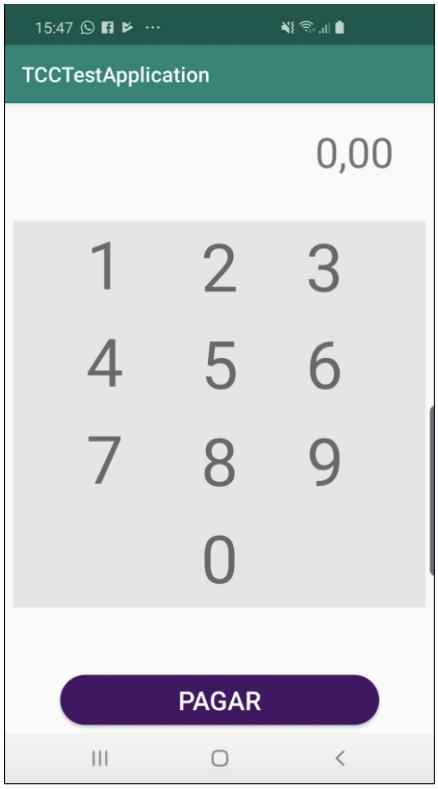

Figura 9: Representação da tela principal do terminal de pagamento desenvolvida para o TCC.

Os tipos de entrega de atualizações para o usuário final se dividem entre entregas de aplicativos de forma individual, distribuídos através de um sistema similar a atualização automática do Google Play e App Store, identificado ao se verificar uma seta para baixo na barra de notificações como podemos verificar na Figura 10, podendo também ver na Figura 11 o detalhe do aplicativo sendo atualizado ao se puxar a barra de notificações.

Figura 10:representação de download de atualização de aplicativos.

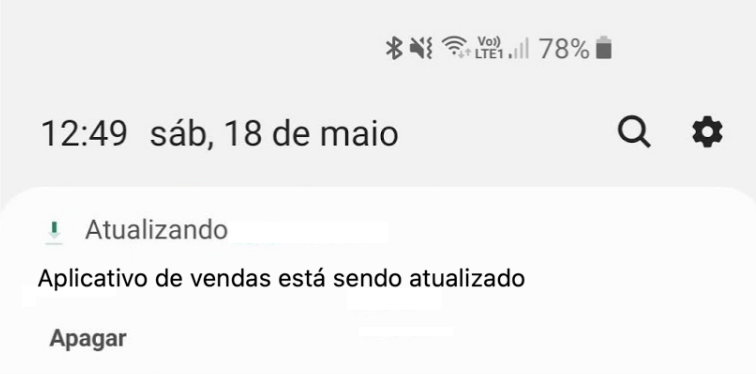

Figura 11:representação de notificação de download de atualização de aplicativos.

Este tipo de atualização tem pouco impacto no funcionamento do dispositivo, além de pouco ou nenhum impacto na usabilidade do mesmo. Temos também entrega de atualização do sistema operacional, quando são justificadas por alterações diretamente no OS, como alteração de *drivers*, ou quando existe alteração em diversos aplicativos com algum nível de interdependência.

Neste caso, a entrega é realizada por um sistema diferente, que atualiza todo o sistema, utilizando o sistema nativo do Android, que necessita da reinicialização do sistema operacional para a aplicação da atualização como representado na Figura 12, mas sem apagar as partições com os dados do usuário, caso contrário funcionaria como uma formatação do sistema operacional, sendo inconveniente para o usuário final.

Figura 12: representação de atualização de sistema android.

O pacote de atualização que contém o sistema operacional atualizado pode conter o sistema inteiro chamado de *Over-the-Air (*OTA) *full*, ou apenas as alterações a partir de uma versão pré-estabelecida, ou OTA *diff* [24].

Os testes realizados a cada *Sprint* inicialmente eram primariamente manuais, seguindo o roteiro da certificação supracitada.

A metodologia de desenvolvimento da empresa é o ágil, com *sprints*, que são ciclos de desenvolvimento no formato de *timebox* [25], que são um contrato de equipe a respeito do tempo designado para se finalizar um objetivo e que usualmente varia entre uma e quatro semanas. No caso da empresa mencionada o Sprint tem duração de 2 semanas, a *Project Owner* (PO), que é o papel da pessoa responsável por gerenciar o *backlog* de atividades referentes ao projeto [2, p. 50], de forma a atingir o objetivo desejado ao final do ciclo de desenvolvimento [21, pp. 125-126], garantindo uma visão compartilhada sobre o direcionamento do projeto de forma a garantir um bom retorno de investimento [26, p. 23].

Neste caso de estudo, a PO prioriza junto ao usuário final as issues mais relevantes para o *sprint* que se encontram no *backlog*, que é uma lista de atividades a serem desenvolvidas para o projeto [2, p. 50] e durante as *plannings*, que é um evento onde as equipes determinam os itens do *backlog* nos quais eles vão trabalhar [21, pp. 285-287] [26, p. 5], as equipes pontuam as *issues* escolhidas usando o *planning poker*, onde os membros da equipe votam com auxílios de cartões com valores, que ficam escondidos até um momento escolhido para revelação simultânea. O processo ocorre na presença da PO e aqueles que tiverem pontuações muito acima ou abaixo da média, argumentam o motivo de suas escolhas e posteriormente, outra rodada pode ser realizada para que se atinja um consenso [26, pp. 56-59].

De acordo com a média de pontos que a equipe costuma entregar, se determina a quantidade de *issues* que vão entrar no próximo *sprint*. Durante o andamento do *sprint*, ocorrem em horários pré-estabelecidos pela equipe rápidas reuniões diárias para manutenção do foco da equipe no trabalho chamadas de *dailies* [21, p. 342], mas também conhecidas como *stand-up meetings* [2, p. 51]. O escopo da reunião é definir no que cada membro da equipe está trabalhando a curto prazo [26, p. 28]. Ao final do *sprint* ocorre a *review*, onde a PO junto da equipe, apresentam as alterações que foram feitas com sucesso durante a *Sprint* para o usuário final e que já estão prontas a serem entregues.

Posteriormente se realiza uma retrospectiva, que é uma reunião que ocorre regularmente onde se tomam decisões baseadas nas experiências recentes para que realizar melhorias ou correções no processo [27]. No exemplo estudado, a retrospectiva ocorre no final de cada Sprint, onde se grifam as coisas que foram boas e as que precisam ser melhoradas no processo e as ações necessárias para tal além das lições aprendidas.

Apesar da entrega de *software* seguir a metodologia ágil, o processo de certificação, que é a versão do usuário final dos testes de aceitação do usuário, ocorrem

em janelas distintas, e apenas após a entrega completa dos requisitos solicitados previamente, sendo mais próximo de uma metodologia cascata, onde se o processo de certificação falha, temos uma correção somente na próxima entrega de versão, com outra janela de certificação sendo solicitada, ao invés da evolução natural do ágil que seria a correção e entrega incremental o mais rápido possível.

Originalmente, todos os testes eram realizados de forma manual, porém, devido a curta duração dos *sprints*, isso significava não ter uma cobertura desejável de testes, principalmente levando-se em consideração que as duas semanas deveriam englobar o desenvolvimento, os ritos ágeis e os testes, funcionais e não-funcionais. Quando eram encontrados *bugs*, ou a entrega era realizada, mesmo com as falhas encontradas, diminuindo a percepção de qualidade, ou muitas horas-extras eram realizadas em prol do cumprimento do prazo. As máquinas utilizadas para a construção dos artefatos entregues eram as próprias dos desenvolvedores. Este cenário era comum, e a entrada em produção de novas funcionalidades era demorada, chegando a dez meses de entregas contínuas sem aprovação para atualização do campo de dispositivos em produção.

Para se solucionar este problema, uma série de melhorias no processo de testes e desenvolvimento foram implementadas. Testes unitários que ocasionalmente desenvolvidos utilizando o método TDD baseados nas funcionalidades criadas ou nos *bugs* que foram corrigidos, como por exemplo, testes para um método que deva verificar se a função crédito deve ou não estar ativa dependendo de determinada informação lida a partir do cartão utilizado no aparelho ao se realizar um pagamento. Os testes unitários são escritos normalmente pelo próprio programador que está codificando a atividade, e a IDE Android Studio normalmente é utilizada para realizar as execuções dos testes unitários durante o desenvolvimento conforme pode ser visto na Figura 13. Visto que é um teste unitário e o mesmo deve somente testar um método sem um cartão real, a informação que alimenta o método é gerada usando o Mockk, visto que o código é primariamente em Kotlin, e saída do método seria se a função crédito estaria dentre as disponíveis no *array* de resposta do método. Os testes unitários são executados a cada

processo de compilação do código, sendo possível avaliar durante as alterações de código, qualquer quebra de contrato entre as diferentes partes do *software*.

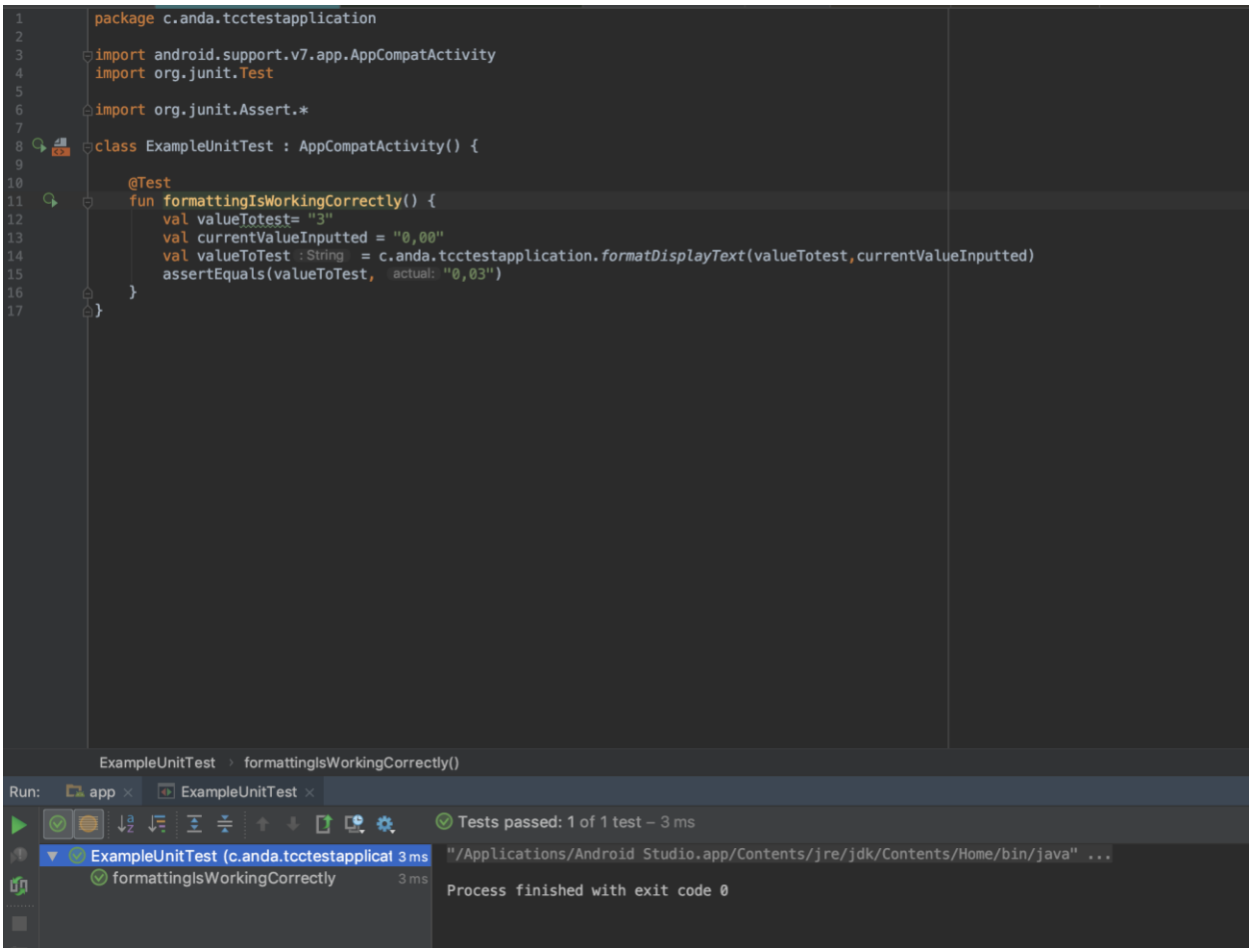

Figura 13: exemplo de teste unitário que valida a formatação do valor demonstrado na tela inicial do software considerando que se enviou para a função o valor 3 e o valor inicial no display de 0,00

Também testes integrados do fluxo em sua totalidade utilizando uma **DSL** – *Domain Specific Language*, que é uma linguagem que tem como objetivo facilitar o entendimento de código de programação por pessoas que estão familiarizadas com o negócio, ou o domínio [28], porém não tem necessariamente experiência em programação, como pessoas da área de negócios por exemplo. Esta DSL foi criada internamente para validar o funcionamento do fluxo de pagamento do início ao fim, utilizando o conceito da aplicação, que foi desenvolvida usando a ideia de máquinas de estado. Com essa DSL, os desenvolvedores a cargo de codificar o teste, configuram os parâmetros de estado e utilizam arquivos com as respostas previamente capturadas do

pinpad, que é a parte do *hardware* responsável pela leitura do cartão e da resposta criptografada de autorização do mesmo, após verificar tanto a validade do mesmo, como a senha inserida, com um cartão real, e as respostas do *host* de pagamento ao se realizar, manualmente o cenário que se deseja automatizar.

O objetivo de tal nível de teste, é dar uma confiabilidade maior ao fluxo que a aplicação deve seguir em dadas ocasiões, e foi programada para seguir os mesmos cenários da certificação anteriormente mencionada. O teste consegue validar as mensagens que seriam, enviadas ao *host* de pagamento, consegue verificar o texto que apareceria em dado momento do fluxo na tela do aparelho, além de garantir a ordem e a aparição do mesmo, em uma comparação mais próxima, seria como as validações do Selenium, porém sem abrir a tela visualmente e com a velocidade de execução de um teste unitário.

No entanto, não é possível cobrir todas as possibilidades com os testes integrados de fluxo, visto que o *host* de pagamento pode mudar o comportamento de determinadas entradas, assim como o *pinpad*, que tem seu comportamento ditado pela carga inicial feita por uma consulta no *host* de pagamento em um procedimento que é chamado de cargas de tabelas de inicialização. Assim como os testes unitários, os testes integrados são executados a cada vez que o código é compilado.

O próximo nível de testes que passou a serem realizados, são testes automatizados de interface, que rodam diretamente no dispositivo. Os mesmos são desenvolvidos utilizando o *framework* Espresso[18], anteriormente utilizando a linguagem Java, e mais recentemente convertidos para Kotlin como pode ser visto no exemplo da Figura 14. Este teste provê um nível de confiabilidade maior, por executar o fluxo no aplicativo rodando no dispositivo, no caso do exemplo da nossa Figura 14, podemos verificar a Figura 15, onde verificávamos em tempo real de execução do teste automatizado no dispositivo, o seu comportamento. Visto que os testes instrumentados rodam diretamente no dispositivo, como simulação de uma interação humana, temos,

[18] https://developer.android.com/training/testing/espresso

portanto, as respostas reais do *pinpad* e do *host* de pagamento. No entanto são bem mais lentos do que os testes anteriores, e tem a restrição de não ter a troca de cartões de pagamento, já que é um procedimento que deve ser feito manualmente. Por esse motivo, são testes mais utilizados para verificar a interface, outros aplicativos que não participem do fluxo de pagamento, e o fluxo de pagamento com um cartão digitado, visto que neste caso não seria necessária a interação humana, porém ainda validaria com sucesso a resposta do *host* de pagamento.

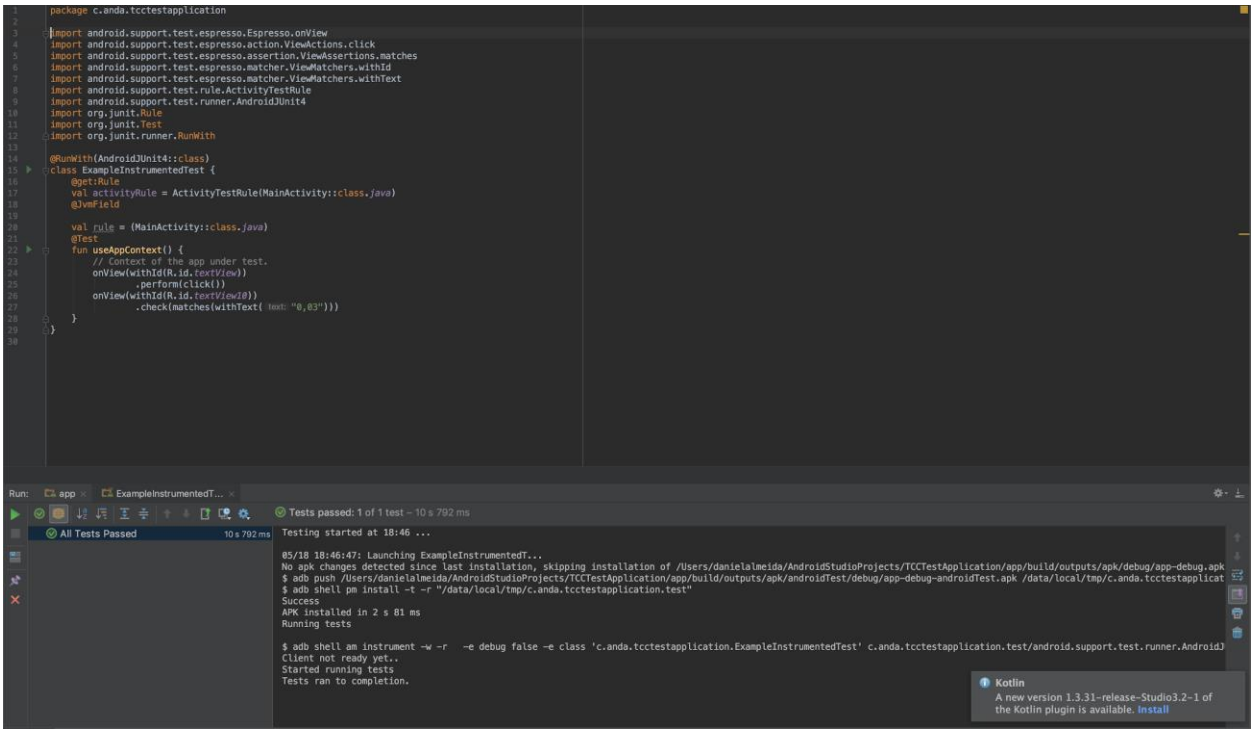

Figura 14: exemplo de teste instrumentado que valida a formatação do valor demonstrado na tela inicial do software considerando que foi selecionado o botão 3 na aplicação.

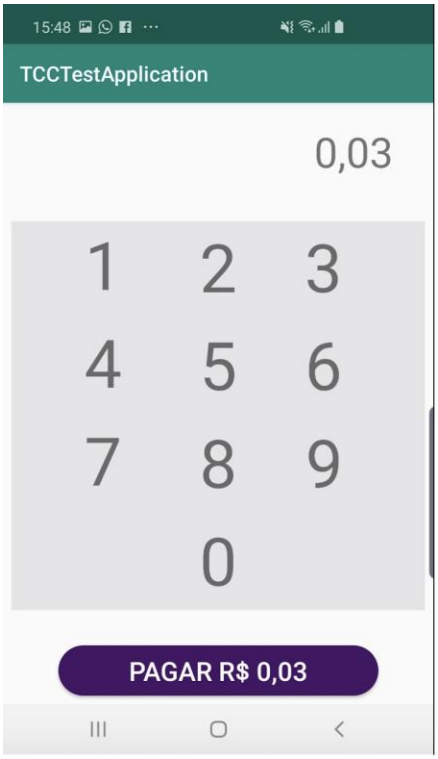

Figura 15: tela do dispositivo durante a execução do teste instrumentado.

Finalmente, os testes manuais, realizados por testadores, que de acordo com a descrição das *issues* abertas para os *bugs* e *features*, validam não só o funcionamento correto do fluxo, mas também os aspectos de usabilidade que não são facilmente identificados em testes automatizados, como por exemplo, a facilidade de se encontrar algum elemento na interface e se o botão que eu devo clicar está visível na tela, além até que nível a interface é amigável para o usuário final. Enquanto os desenvolvedores trabalham na alteração que lhes foi designada, os testadores concorrentemente desenvolvem o teste automatizado com a DSL apresentada anteriormente, após a alteração de código ser compilada em um arquivo APK, O teste é realizado manualmente para as verificações descritas, e o aplicativo e alteração são aprovados para a certificação do usuário final. Caso a entrega seja de OS, também é necessário testar se o OTA funciona adequadamente a partir das versões para a qual um OTA *diff* foi gerado, assim como um OTA *full* para uma versão para qual o *diff* não foi gerado, assim como um teste de OTA para verificar se a versão do OS sendo entregue é passível de atualização futura. Visto que o aplicativo de atualização de sistema verifica se existe

alguma atualização validando pelo código da versão em ordenação crescente, comparando com a atual, existe sempre um OS, com código idêntico ao mais atual, com o código da versão "999999" para a validação deste caso. A entrega de uma versão do sistema operacional que não é passível de futuras atualizações é o pior caso possível, já que seria necessário o *recall* de todos os dispositivos, para atualização manual para uma versão que pudesse novamente sofrer alterações entregues pela internet.

O controle de versionamento de código é utilizando Bitbucket da Atlassian, mesma empresa do Jira, usando o GIT. Temos um servidor na AWS para builds de apps e outro para builds do OS com Jenkins que automaticamente realiza uma varredura na conta do Bitbucket do projeto buscando por novas *branches* ou alterações nas existentes a cada uma quantidade de tempo estabelecida, assim como sob demanda, de forma manual. Caso seja encontrada alguma nova versão de código, o *checkout* da mesma é realizada, a configuração das variáveis de ambiente, a limpeza de cache das dependências do projeto, a compilação do mesmo é realizada, os passos de teste integrado e instrumentados criados com a DSL, criação de uma *tag* com o intuito de congelar as alterações realizadas em uma dada versão para posteriores *bugfixes*, e integração ao repositório de apps no repositório de aplicativos que formam o OS, além do cadastro no servidor que guarda as versões de aplicativos existentes caso o número da versão tenha sido incrementado em uma *branch* específica de *release candidates* que é configurada na varredura feita pelo Jenkins. Podemos verificar um exemplo de um build sendo realizado e a execução de seus testes automatizados via Jenkins na Figura 16.

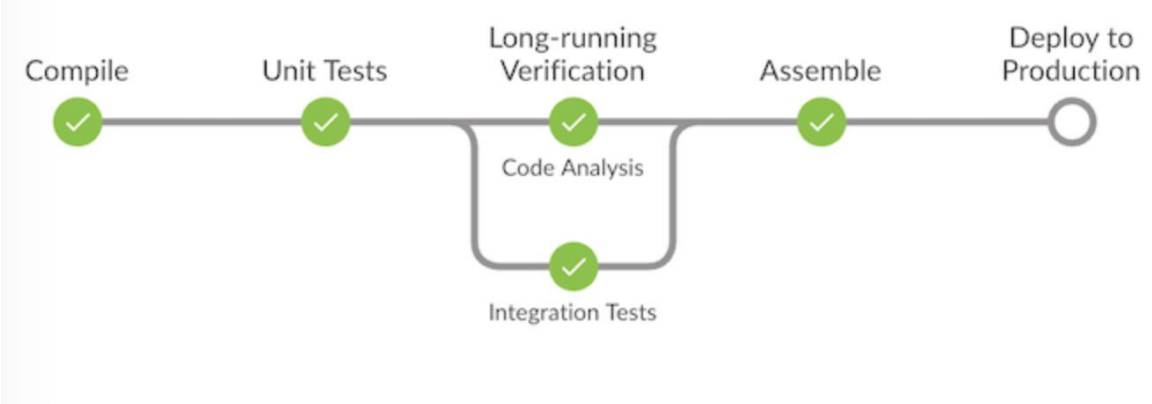

Figura 16: Exemplo de execução de steps automatizados com integração contínua [29]

Após as melhorias descritas, as entregas que de fato são aplicadas em produção passaram a ser muito mais frequentes, caindo de dez meses para aproximadamente dois, além da diminuição drástica dos *bugs* e sua gravidade encontrados no procedimento de aceite da certificação, mesmo quando não é aprovado imediatamente. A percepção de qualidade, para o cliente final aumentou, gerando maior confiança na equipe de desenvolvimento e a qualidade do trabalho da equipe de testes também melhorou, visto que as automações sendo desenvolvidas pela equipe de qualidade, incrementa o conhecimento em linguagens de programação, mesmo que os profissionais originalmente possuam outros *backgrounds* profissionais, além de desafios diferentes e menos repetitivos. A quantidade de vezes que foi virar noites para realizar entregas no último segundo possível também foi reduzido a quase zero, visto que o procedimento de *build* automático é muito mais rápido e livre de erros que o realizado anteriormente na máquina do desenvolvedor, e a cobertura de boa parte dos testes anteriores nos níveis apresentados, garante uma menor possibilidade de um erro inesperado, gerando menos retrabalho.

3.3 DISCUSSÃO SOBRE OS CASOS

No caso de estudos 1 a empresa de telecomunicações, mesmo usando o modelo cascata, com uma fase de testes bem definida, geralmente tinha que alocar uma quantidade de tempo para os processos de testes. A criação de procedimentos e ferramentas, acelerou a execução do trabalho dos testadores, de maneira a ser possível uma maior cobertura na mesma quantidade de tempo e o uso do tempo excedente para a realização de mais melhorias nos processos por vias da criação de mais automações, além do desenvolvimento pessoal dos profissionais em diversas outras áreas como a de gestão.

No caso de estudos 2 a empresa de pagamentos teve uma clara melhoria com a implantação dos procedimentos de automatização de processos e de testes realizados em relação aos procedimentos manuais. Apesar do tempo de *sprints* não ter sido alterado antes e depois da melhoria, vimos que a partir do aumento da qualidade de um produto, tivermos maior confiança do cliente no produto.

Tal confiança foi percebida com atualizações aparecendo no produto final em campo com mais frequência, e isto, por consequência aumentou o grau de satisfação da equipe de desenvolvimento com o cumprimento de seus objetivos e aumentando o engajamento da equipe testadora com linguagens de programação.

Também se chegou ao entendimento que ao contrário de ser uma substituição da sua força de trabalho, o engajamento com esse tipo de experiência torna o profissional mais diferenciado frente ao mercado, além de facilitar o trabalho intelectual com um crivo mais exigente para questões que muitas vezes eram deixadas de lado devido a finalização dos prazos.

Em ambos os casos a economia de esforço gerada ao profissional, se retroalimenta na forma do desenvolvimento de novas melhorias que geram maior qualidade com menos esforço futuro.

4 CONCLUSÃO

Conforme estudado, tanto nos modelos cascata quanto ágil apresentados nos estudos, a automação implementada na fase de teste de um projeto provê as esperadas melhorias de economia, mitigando a possibilidade de surpresas de última hora nas fases finais de implantação de um projeto em produção, mas acima de tudo garantindo um nível de qualidade que mantenha a satisfação do usuário final, garantindo um bom relacionamento com a organização desenvolvedora e seus profissionais que ao serem removidos de um ciclo contínuo de entregas apressadas conseguem se desenvolver melhor tecnicamente e adicionar novas melhorias ao processo de forma a criar um novo ciclo, mas um de melhoria constante da qualidade e confiança no produto, e conseguem focar o seu trabalho em questões onde a automação não substitui o trabalho humano, como análises subjetivas e experiência de uso.

Mesmo que a metodologia e tamanho da equipe variem, o gargalo evidenciado por processos de testes ineficazes nos apontou diretamente onde as melhorias traziam um benefício mais imediato.

Nos casos de estudo, avaliamos que o uso de automações tanto para auxílio da execução do teste manual, como para manutenção contínua da qualidade entre alterações do software, aceleram o processo de desenvolvimento como um todo, visto que a fase de testes é essencial a qualquer entrega de produto que será submetido por qualquer formato de teste de aceitação do usuário final, assim como aumentam a confiança no produto entregue, como mencionado no caso de estudo 3.2, que a frequência de entregas aumentou drasticamente.

Para toda a equipe envolvida no desenvolvimento a consequência é um aumento na satisfação, ao conseguir enxergar o seu produto sendo utilizado em estabelecimentos, ou sendo anunciado em promoções nos meios de comunicação, mostrando que a automação é um investimento que usualmente vale a pena para todos os tamanhos de empresa.

4.1 TRABALHOS FUTUROS

O interesse de alguns membros mais leigos da equipe de testes provê um interessante trabalho a ser desenvolvido futuramente para o ensino de linguagem de programação utilizando DSLs, conforme citado na Seção 3.2, para simplificação de alguns códigos e o resultado imediato dos mesmos em testes instrumentados.

Visto que as verificações de testes instrumentados, são em sua maioria simples, assim como seus comandos. A criação de métodos que traduzem para um ambiente específico, que seja familiar a quem deseja o aprendizado simplificaria um contato inicial com as ferramentas e a lógica de programação, e com isso desejar-se-ia verificar um incremento na capacidade técnica geral de todos os interessados envolvidos na equipe, independente de sua bagagem tecnológica e profissional, como um exemplo prático, uma PO.

Nas experiencias relatadas, alguns membros da equipe de testes, apesar de estarem a muitos anos no mercado, não tinham contato com programação diretamente por muito tempo, já que o exigido destes profissionais era um teste manual, do tipo caixa-preta, ou onde a execução de comandos para verificação de algum resultado seguia um roteiro específico.

No entanto, a direção do mercado para o uso de métodos ágeis nos últimos tempos tem exigido cada vez mais um profissional de testes que seja um membro ativo da equipe de desenvolvimento em atividades de automação, e a implantação das soluções apresentadas foi o estímulo necessário para fomentar o interesse no envolvimento com automação.

Outro trabalho a ser desenvolvido é um estudo a respeito dos benefícios da programação em pares, onde dois programadores trabalham juntos em uma única estação de trabalho para a qualidade do software [30]. Além da melhoria da qualidade,

poderia se medir o nível de conhecimento obtido com a cooperação entre profissionais e a melhoria da relação interpessoal entre os mesmos com o companheirismo gerado.

Também poderia ser desenvolvido um estudo onde a rotatividade dos desenvolvedores em uma atividade de teste, preferencialmente a ser desenvolvida pelo testador da equipe, com o intuito de melhorar a qualidade do código ao se gerar uma consciência de testador nos desenvolvedores, enquanto se aumenta a fluência do testador no código trabalhado, facilitando a descrição de novos problemas e análises de defeitos descobertos.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] IEEE Computer Society, “SWEBOK,” 2019. [Online]. Available: https://www.computer.org/education/bodies-of-knowledge/software-engineering. [Acesso em 18 Maio 2019].

[2] I. Sommerville, Engenharia de Software - 9ª Edição, Pearson Education do Brasil, 2011.

[3] J. Tian, Software Quality Engeneering: Testing, Quality Assurance and Quantifiable Improvement, Wiley, 2005.

[4] M. Pezzé e M. Young, *Software Testing and Analysis: Process, Principles and Techniques,* Wiley, 2007, p. 4.

[5] M. Newman, “Software Errors Cost U.S. Economy $59.5 Billion Annually,” 28 Junho 2002. [Online]. Available: https://web.archive.org/web/20090610052743/http:/www.nist.gov/public_affairs/releases/n02-10.htm. [Acesso em 18 Maio 2018].

[6] NASA Johnson Space Center, *Error Cost Escalation Through the Project Life Cycle,* 1981, p. 2.

[7] G. Myers, The Art of Software Testing - Second Edition, Hoboken: John Wiley & Sons, 2004.

[8] C. Symons, *Function point analysis: difficulties and improvements,* IEEE, 1988, pp. 2-111.

[9] J. N. Pereira, *Guia de Apoio de Análise por Ponto de Função: Uma Proposta Preliminar para Interpretação de Processo Elementar,* Brasilia: Universidade Católica de Brasilia, 2014.

[10] T. J. McCabe, *A Complexity Measure,* IEEE Transactions on Software Engeneering, VOL. SE-2, NO.4, 1976.

[11] mds.br, “Conceito: Teste de Aceitação,” 2006. [Online]. Available: http://mds.cultura.gov.br/core.base_rup/guidances/concepts/acceptance_testing_12A0F152.html. [Acesso em 18 Maio 2019].

[12] Google Developers, “Releasing Your Actions to Alpha and Beta Environments,” 12 Março 2019. [Online]. Available: https://developers.google.com/actions/deploy/release-environments. [Acesso em 18 Maio 2019].

[13] PCMAG, “Definition of: release candidate,” [Online]. Available: https://www.pcmag.com/encyclopedia/term/50384/release-candidate. [Acesso em 18 05 2019].

[14] K. Naik e P. tripathy, *Software Testing and Quality Assurance: Theory and Practice.,* Wiley, 2008.

[15] F. Tsui e O. Karam, *Fundamentos de Engenharia de Software.,* LTC, 2013, pp. 146-148.

[16] K. Beck, *Extreme Programming Examined,* Boston: Addison-Wesley, 2001.

[17] M. Fowler, “Continuous Integration,” 01 Maio 2006. [Online]. Available: https://martinfowler.com/articles/continuousIntegration.html. [Acesso em 18 Maio 2019].

[18] H. D. Bennington, *Production of Large Computer Programs,* 1983.

[19] W. Royce, “Managing the Development of Large Software Systems,” em *Managing the Development of Large Software Systems*, 1970, pp. 1-9.

[20] Agile Alliance, “User Stories,” Agile Alliance, [Online]. Available: https://www.agilealliance.org/glossary/heartbeatretro/. [Acesso em 18 05 2019].

[21] M. Cohn, Succeding with Agile: Software Development Using Scrum, Addison-Wesley Professional, 2009.

[22] D. Huizinga e A. Kolawa, *Automated Defect Prevention: Best Practices in Software Management,* John Wiley & Sons, 2007, p. 11.

[23] C. Karat, C. Halverson, D. Horn e J. Karat, Patterns of entry and correction in large vocabulary continuous speech recognition systems, CHI 99 Conference Proceedings, 1999, pp. 568-575.

[24] Android Source, “Reducing OTA Size,” 2019. [Online]. Available: https://source.android.com/devices/tech/ota/reduce_size. [Acesso em 18 Maio 2019].

[25] Agile Alliance, “Iteration,” Agile Alliance, [Online]. Available: https://www.agilealliance.org/glossary/iteration. [Acesso em 18 05 2019].

[26] M. Cohn, Agile Estimating and Planning, Upper Saddle River: Pearson Education, 2006.

[27] Agile Alliance, “Heartbeat Retrospective,” Agile Alliance, [Online]. Available: https://www.agilealliance.org/glossary/heartbeatretro/. [Acesso em 18 05 2019].

[28] M. Fowler, Domain-Specific Languages, Addison-Wesley Professional, 2011, pp. 27-28.

[29] bmuschko.com, “Build Pipelines with Jenkins 2 by example,” 2017. [Online]. Available: https://bmuschko.com/blog/jenkins-build-pipeline/. [Acesso em 18 05 2019].

[30] L. Williams, *Integrating pair programming into a software development process,* Chartlotte: 14th conference on Softeare Engeneering Education and Training., 2001, pp. 27-36.